SEMGREP

FROM ZERO TO HERO

AGENDA

TOPICS ~2h

- Intro into Semgrep
- Creation of custom rules
- CWE vs CVEs

LABS ~2h

- First contact with the Semgrep CLI
- Writing a custom rule
- Understanding taint flow
- Mob exercise - assessing your code together

~1h Lunch break

What is

this Semgrep thingy ...

History: SGREP (Syntactic GREP)

- Initially called Sgrep/Pfff
- Written By Yoann Padioleau at Facebook for analyzing PHP code
- Was used to Enforce Best Practices
- Easy for developers to organize and understand the rules
- Joined R2C and renamed Sgrep to Semgrep
- Goal was to match based on semantics of the code

Source: https//semgrep.dev/blog'2021/semgrep-a-static-analysis-journey

SEMGREP IN A NUTSHELL

Fast and lightweight static analysis tool to find bugs and enforce code standards.

SEMGREP IN A NUTSHELL

SAST

Sometimes also Static Analysis

Static Application Security Testing

Goal: Find (hidden) vulnerability by reviewing your source code

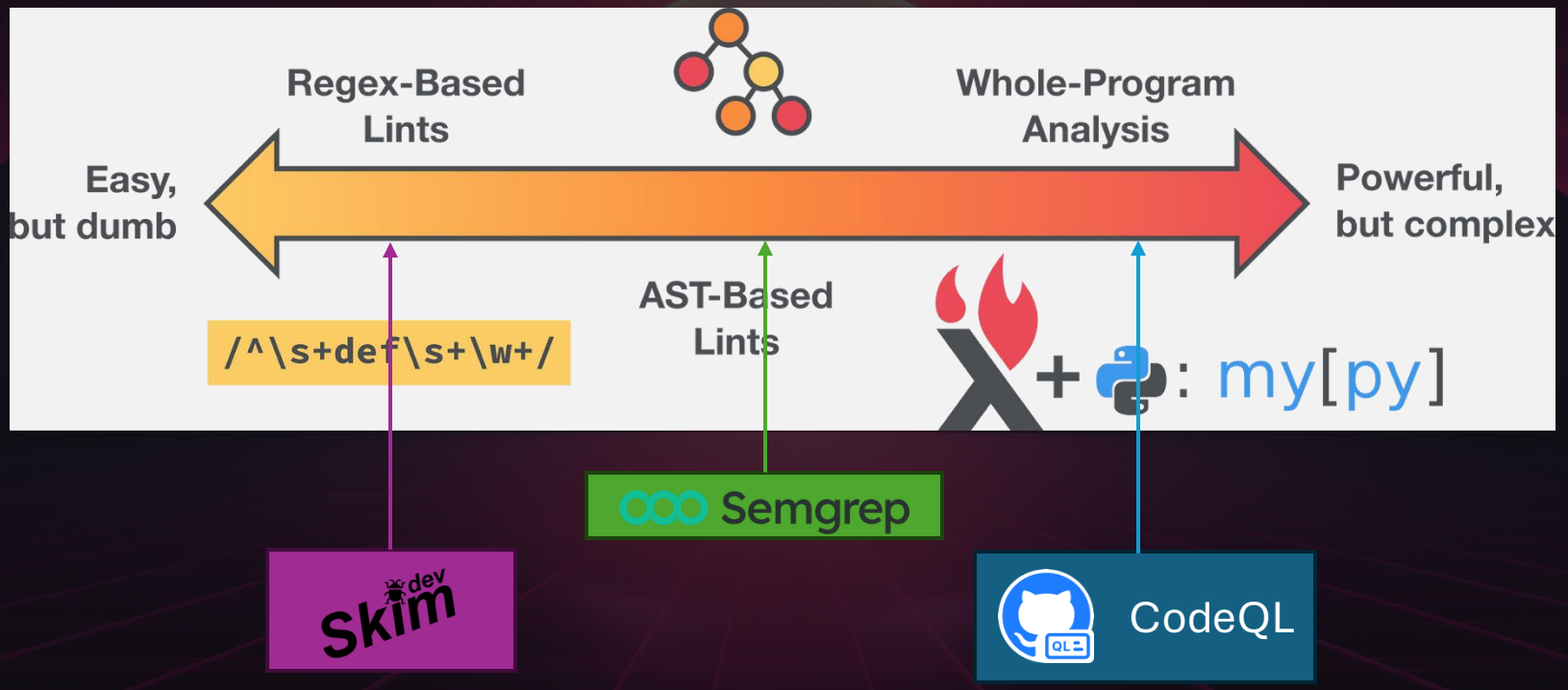
- Detection of flaws in an early stage
- Aids code reviews through automation
- Covers your whole codebase
- Can find injection flaws, deserialization, XSS, (Security) code smells and more

BUT

- Tons of output that needs to be assessed
- Sometimes a lot of false positives (fuzzy logic isn't the smartest)
- No replacement for Pentest or Dynamic Analysis
- Exotic coding languages might not be covered

SEMGREP IN A NUTSHELL

SAST Slider



Source: https://instagram-engineering.com/static analysis-at-scale-an-instagram-story-8f498ab71a0c

Why shall

I care about

Semgrep

?

WHY SEMGREP?

Problem: Find disabled SSL certificate validation in Python

Requests can also ignore verifying the SSL certificate if you set `verify` to False:

```
>>> requests.get('https://kennethreitz.org', verify=False)
<Response [200]>
```

Note that when `verify` is set to `False`, requests will accept any TLS certificate presented by the server, and will ignore hostname mismatches and/or expired certificates, which will make your application vulnerable to man-in-the-middle (MitM) attacks. Setting verify to `False` may be useful during local development or testing.

Source: https://requests.readthedocs.io/en/latest/user/advanced/

WHY SEMGREP?

Let's use grep

```
import requests

def request1():
  return requests.get('https://securepayments.com/get-cc-data‘, verify=False)
```

```
$ grep “verify=False” request_problem.py
```

WHY SEMGREP?

Let's use grep

```
import requests

def request2():
  return requests.get('https://securepayments.com/get-cc-data', verify = False)
```

```
$ grep “verify *= *False” request_problem.py
```

WHY SEMGREP?

Let's use grep

```
import requests

def request3():
  VERY_TRUE = False
  r = requests.get('https://securepayments.com/get-cc-data',verify=VERY_TRUE)
  return r
```

```
$ grep "verify=????" request_problem.py
```

WHY SEMGREP?

Let’s use grep


```
import reques

def request3(
  VERY_TRUE
  r = requests
  return r
```

```
$ grep “veri
```

Home
Questions
Tags
Users
Companies
LABS
Jobs
Discussions
COLLECTIVES
Communities for your favorite technologies.
Explore all Collectives
TEAMS
overflow AI
Now available on Stack Overflow for Teams! AI features where you work: search, IDE, and chat.
Learn more
Explore Teams

RegEx match open tags except XHTML self-contained tags

Asked 14 years, 11 months ago Modified 8 months ago Viewed 3.9m times

2291

Locked. Comments on this question have been disabled, but it is still accepting new answers and other interactions. Learn more.

I need to match all of these opening tags:

```
<p>
<a href="foo">
```

But not self-closing tags:

```
<br />
<hr class="foo" />
```

I came up with this and wanted to make sure I've got it right. I am only capturing the `a-z`.

```
<([a-z]+) *[^/]*?>
```

I believe it says:

- Find a less-than, then
- Find (and capture) a-z one or more times, then
- Find zero or more spaces, then
- Find any character zero or more times, greedy, except `/`, then
- Find a greater-than

Do I have that right? And more importantly, what do you think?

html regex xhtml

4397

Locked. There are disputes about this answer's content being resolved at this time. It is not currently accepting new interactions.

You can't parse [X]HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML. Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions. Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes. HTML-plus-regexp will liquify the nerves of the sentient whilst you observe, your psyche withering in the onslaught of horror. Regex-based HTML parsers are the cancer that is killing StackOverflow *it is too late it is too late we cannot be saved* the transgression of a child ensures regex will consume all living tissue (except for HTML which it cannot, as previously prophesied) *dear lord help us how can anyone survive this scourge* using regex to parse HTML has doomed humanity to an eternity of dread torture and security holes *using regex* as a tool to process HTML establishes a breach *between this world* and the dread realm of corrupt entities (like SGML entities, but *more corrupt) a mere glimpse* of the world of regex **parsers for HTML will ins**tantly transport a *programmer's consciousness i*nto a world of ceaseless screaming, he comes, ~~the pestilent s~~lithy regex-infection will **devour your HT**ML parser, application and existence for all time like Visual Basic only worse *he comes he comes do not fight* h**e comes, hi**s unholy radiance *destroying all enlightenment, HTML tags **leaking from your eyes like liq**uid* pain, the song of regular expression parsing will ext*inguish the voices of mortal man from the sp*here *I can see it can you see it it is beautiful* the f`inal snuf`fing *of the lies of Man ALL IS LOST* **ALL IS** LOST the *pony he comes* he comes ~~he comes~~ **the ich**or permeates *all* MY FACE *MY FACE* ~~h~~ *god* **no NO NOOO N**O stop *the an*`gles .are` **not real ZALGO IS TONY THE PONY HE COMES**

Have you tried using an XML parser instead?

stackoverflow About Products For Teams Search...

Home

PUBLIC

Questions

Tags

Users

Companies

COLLECTIVES

Explore

TEAMS

Stack Overflow Teams – Start collaborating and sharing organizational knowledge.

Free

Create a free Team

Why Teams?

RegEx match open tags except XHTML self-contained tags

Asked 13 years, 5 months ago Modified 1 month ago Viewed 3

I need to

Answers

oldest newest votes

You can't par

can be used

times before

insufficiently

language ar

down HTML

regular exp

crack. HTM

Jon Skeet

egular exp

arsing H

e love,

ML tog

se HT

uman

L-pl

aug

pt

Regex is not a tool that

destroying all

pression parsing will extinguish the voices

beautiful the final snuf fing of the lies of Man ALL IS LOST

comes he:comes the ich or permeates all MY FACE MY FACE oh god no NO NOOOO NO

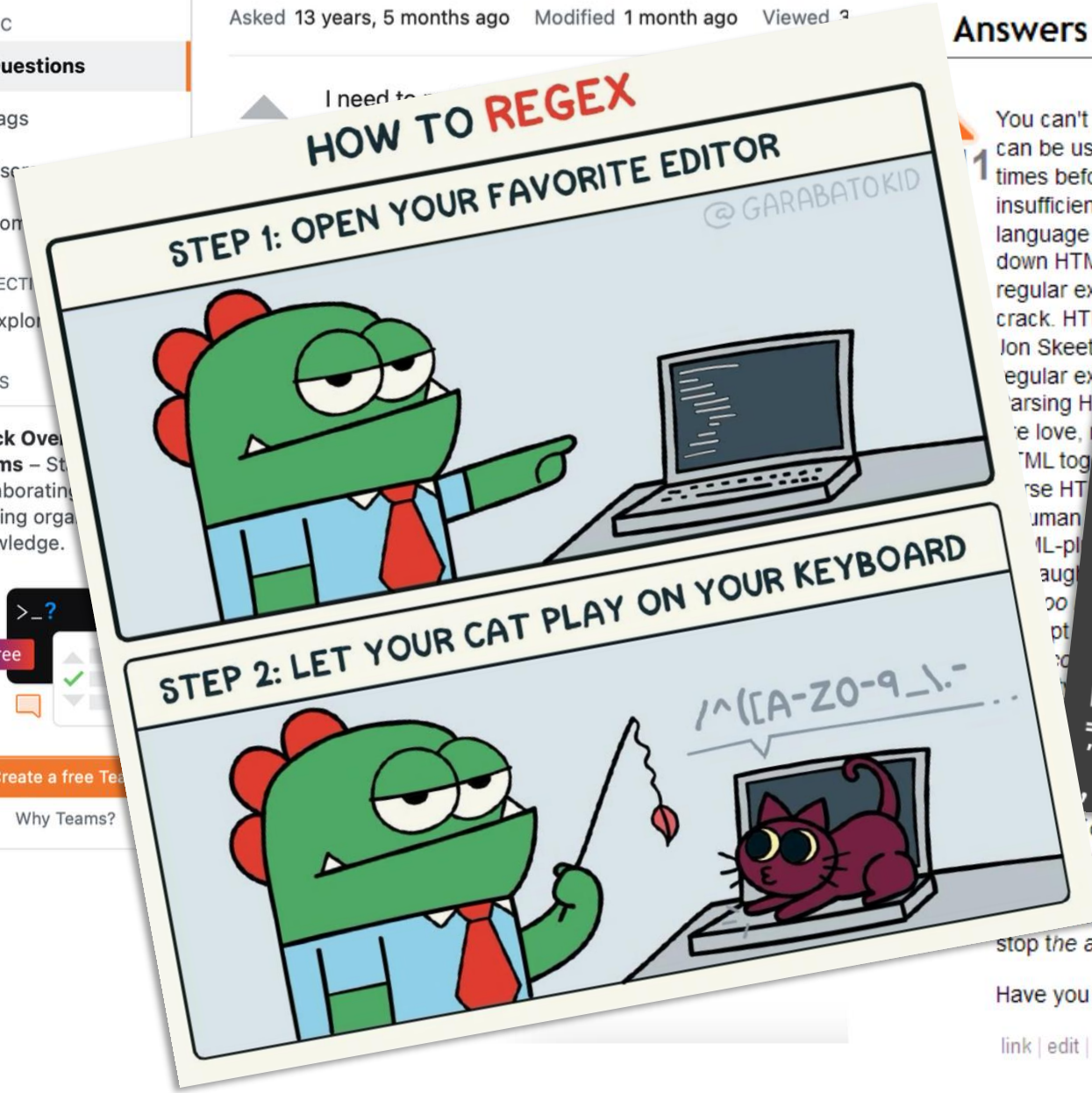
stop the angles are not real ZALGO IS TONY THE PONY, HE COMES

Have you tried using an XML parser instead?

link | edit | flag

edited Nov 14 at 0:18 community wiki

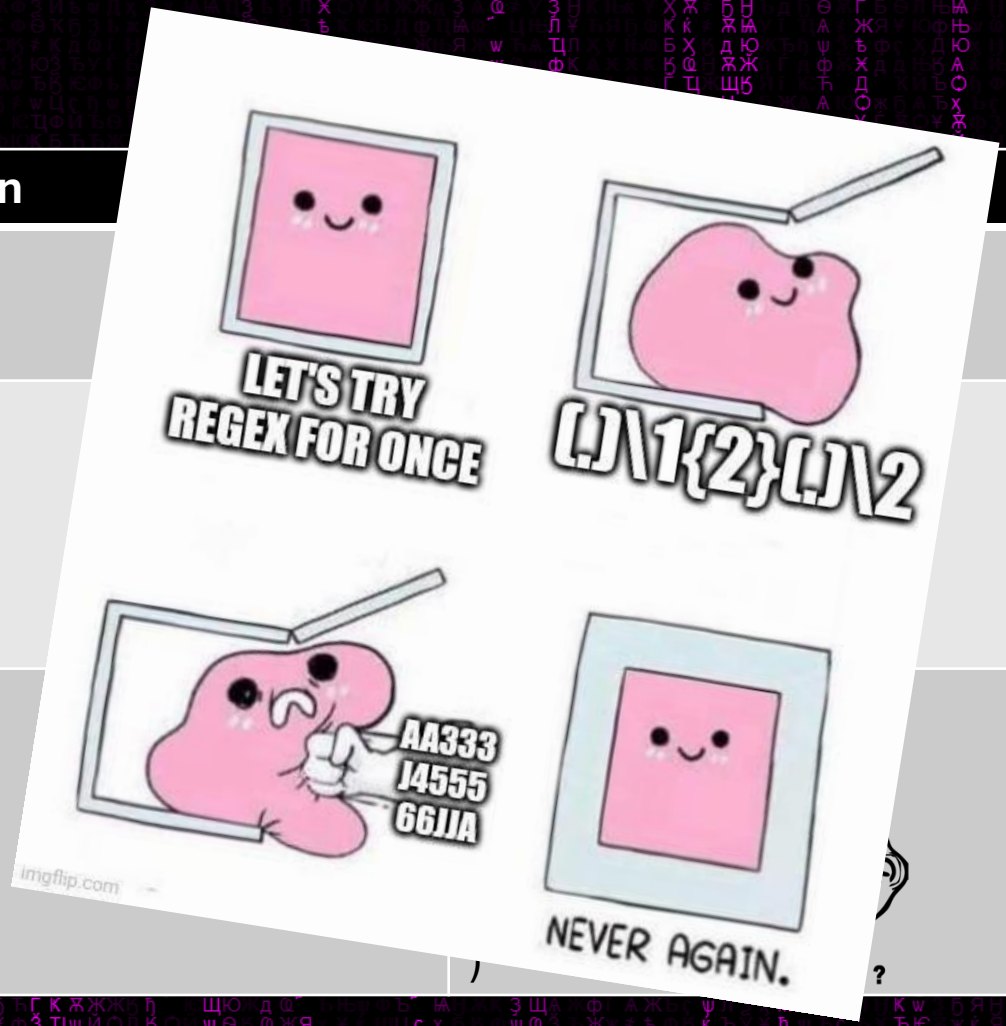
bobince



Regular Expression	Test String
if\(.+\)	if(somecondition) ✓ if (somecondition) ✗
if\s*\(.+\)	if(somecondition) ✓ if (somecondition) ✓

if(a == b)	if(a==b) ✓
	if(
	a==b
	) ✗ 

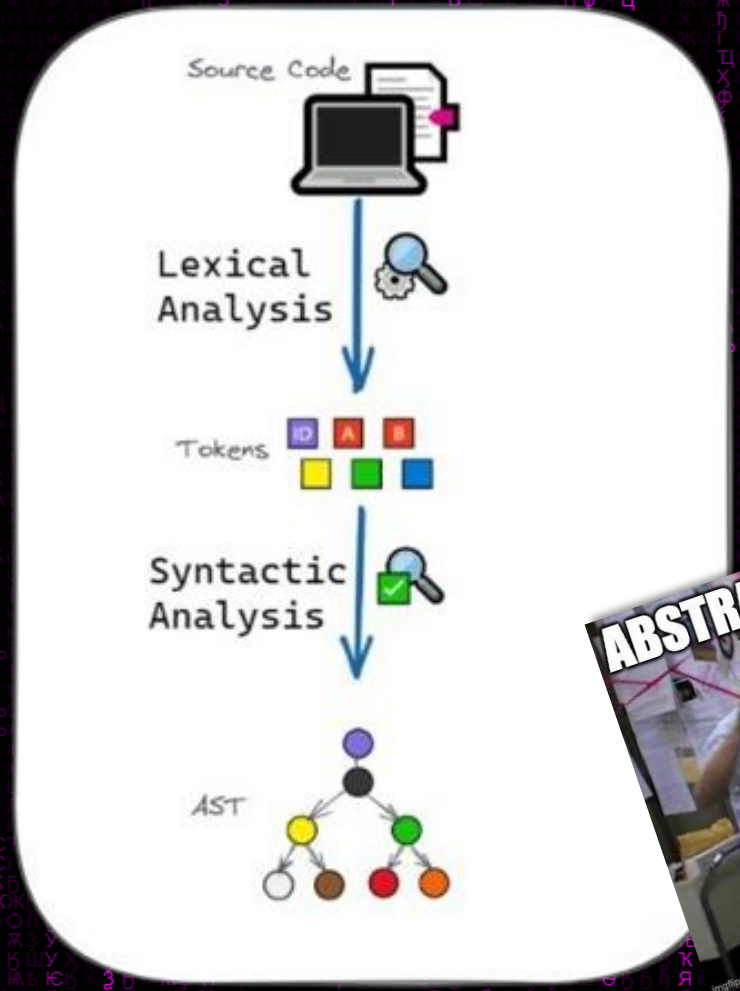
Regular Expression
if\(.+\)
if\s*\(.+\)
if(a == b)

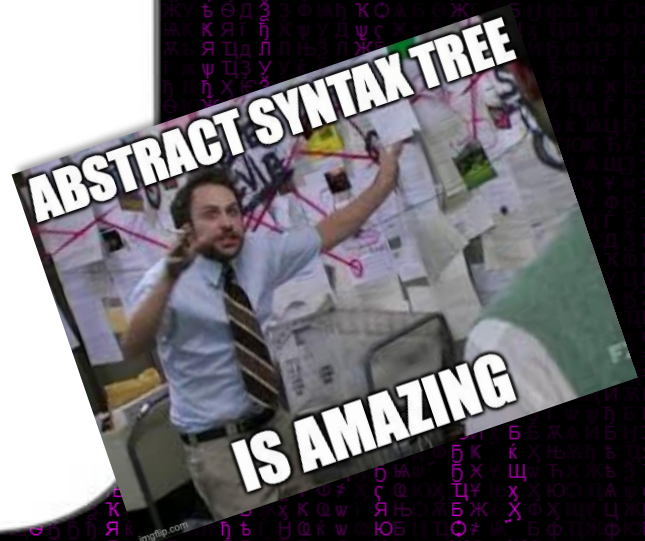


CODE IS NOT A STRING, IT'S A TREE

String != 🌲 Tree

ABSTRACT SYNTAX TREE





An **Abstract Syntax Tree (AST)** abstracts away certain details and retains just enough information to help the compiler understand the structure of the code.

CUTTING OUT THE NOISE WITH SEMGREP TYPED METAVARIABLES

Typed metavariable in Semgrep is a placeholder that only matches code elements of a specified type, enhancing pattern precision by filtering irrelevant matches.

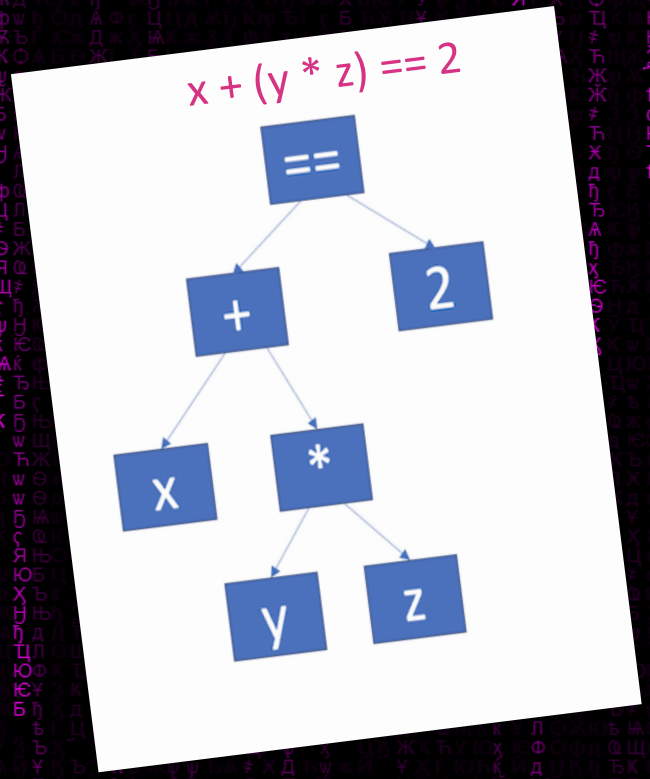
How It Works:

- Remembers types for local variables and arguments to match specific code elements
- Semgrep analyzes code as an AST, not just plain text.

➔Regex in comparison matches on plain text, that is often leading to false positives or missed matches in complex code.

Advantages:

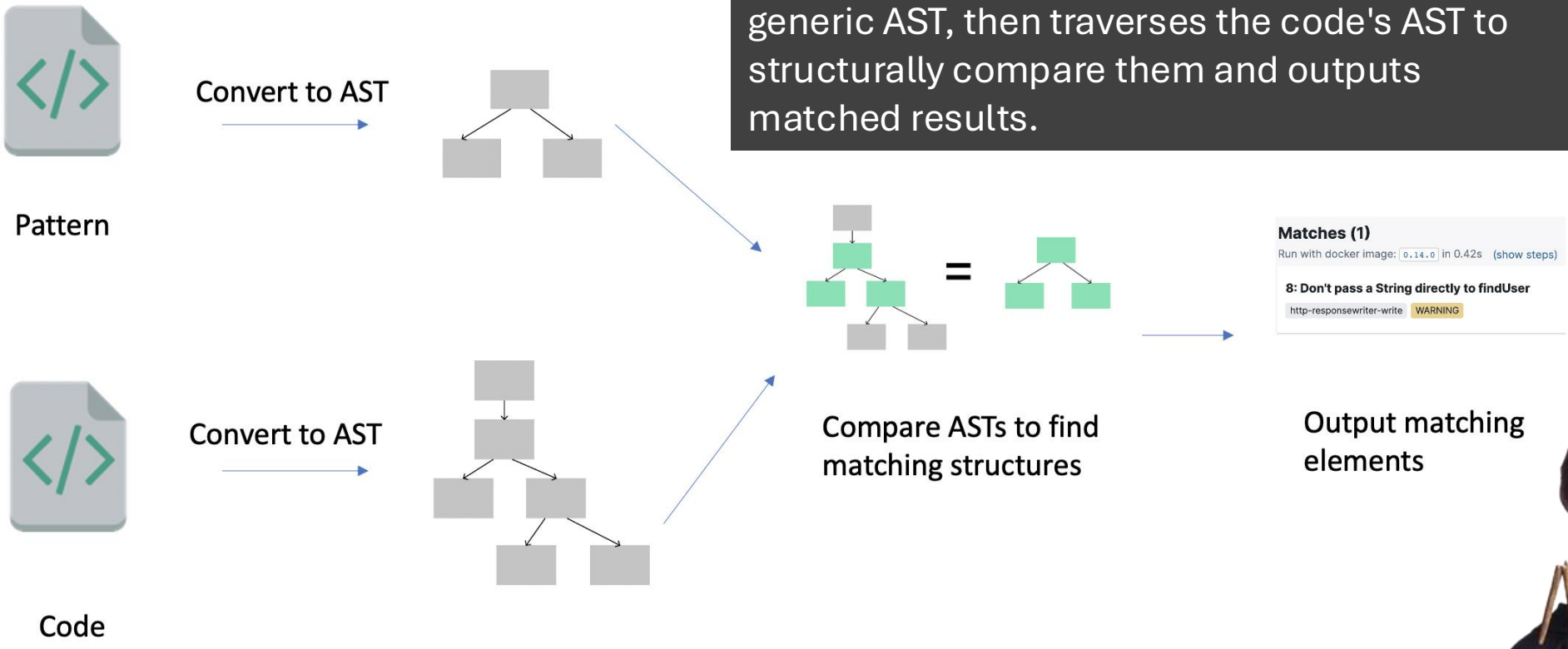
- Ensures matches are structurally and semantically correct, reducing false positives



Source: https://semgrep.dev/blog/2020/type-awareness-in-semantic-grep/

REVEALING THE MAGIC

Semgrep parses the pattern and code into a generic AST, then traverses the code's AST to structurally compare them and outputs matched results.



Source: https://semgrep.dev/blog/2020/type-awareness-in-semantic-grep/

SEMGREP TO THE RESCUE

```
~ bat request_problem.py
       File: request_problem.py
   1   import requests
   2
   3   def request3():
   4       VERY_TRUE = False
   5       r = requests.get('https://securepayments.com/get-cc-data', verify=VERY_TRUE)
   6       return r

~
~ semgrep -e "requests.get(..., verify=False)" -l py request_problem.py


 1 Code Finding

    request_problem.py
           5┆ r = requests.get('https://securepayments.com/get-cc-data', verify=VERY_TRUE)


 Scan Summary

Ran 1 rule on 1 file: 1 finding.
~
```

LABI: FIRST CONTACT

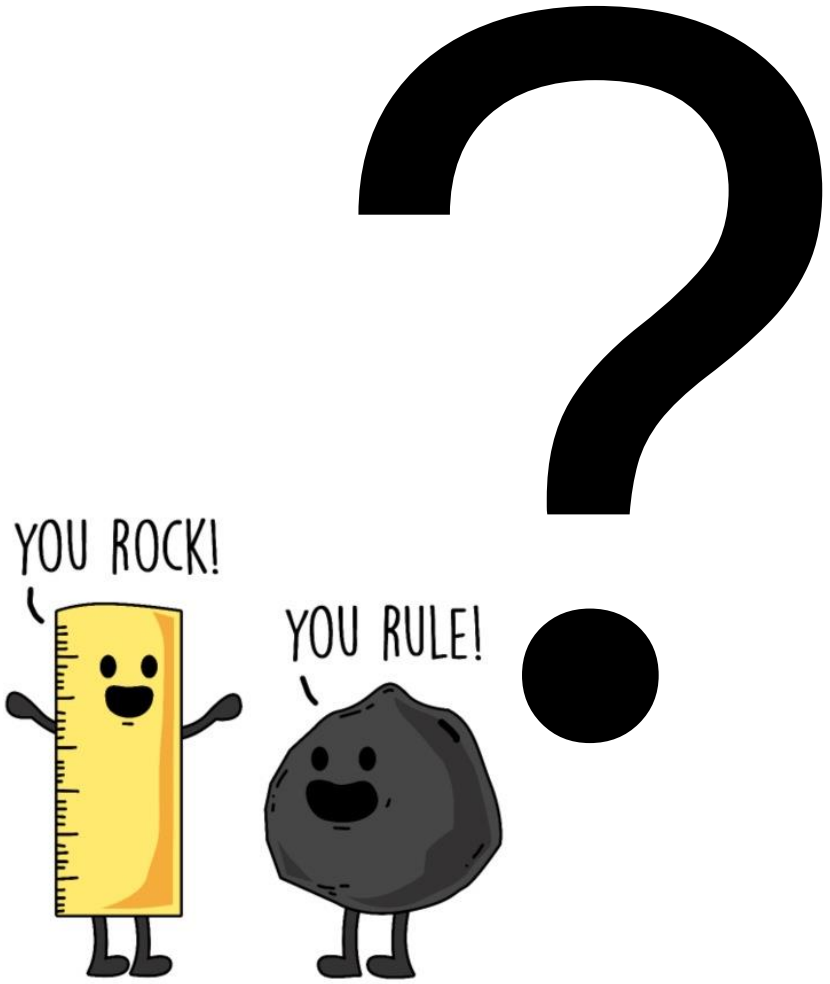
Time estimation: 20min

Topics:

- Exploring the CLI
- CI vs Scan
- First scan
- Ignored rules

Where are

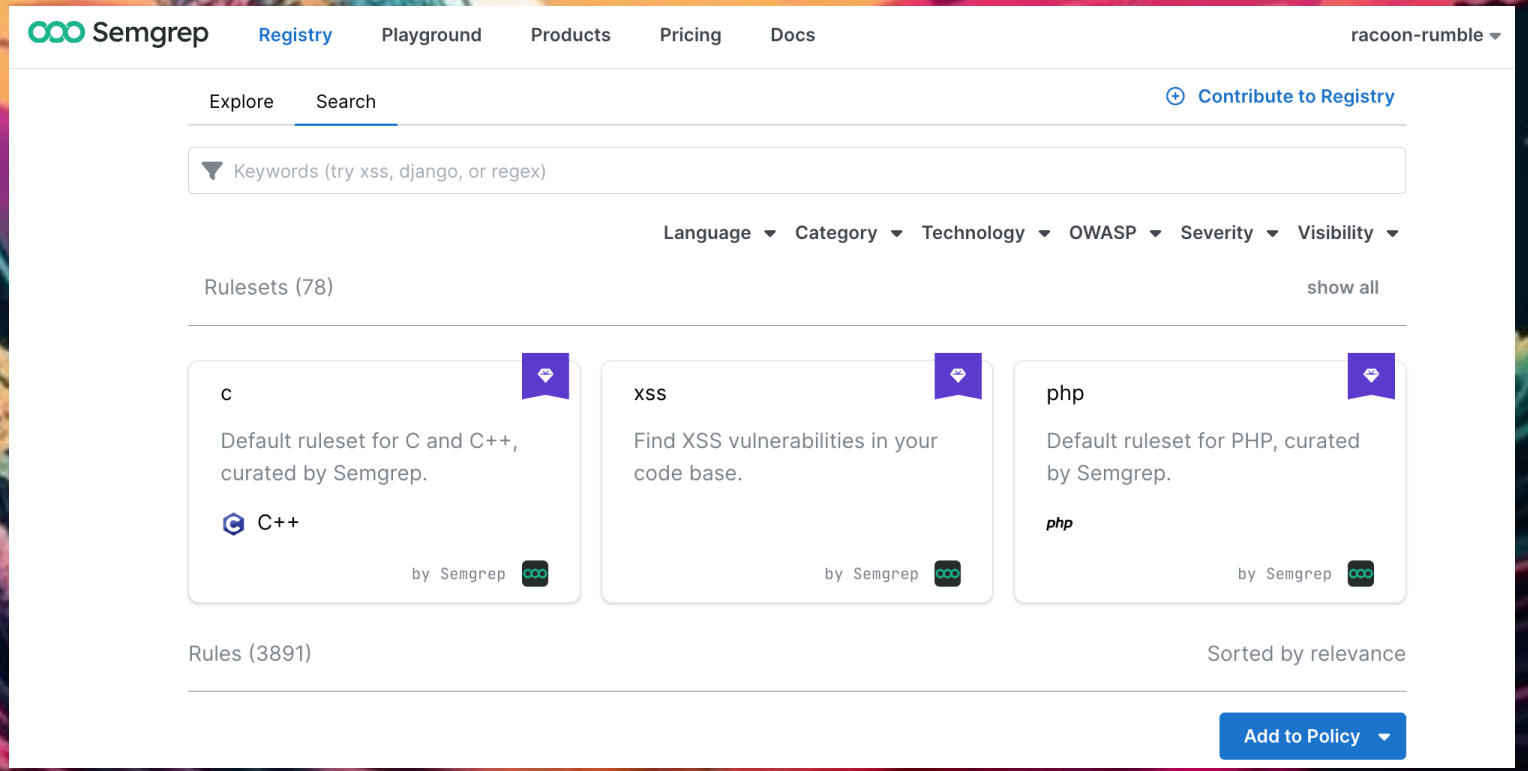
those rules...



SEMGREP REGISTRY

- >3.8k rules Publicly available
- More available for Pro licence
- Support for 20+ Languages
- $ semgrep –config=auto

Hint: You can interact with a big variety of rules in the Playground (you will have chances to see this in the labs)



Source: https://semgrep.dev/r

SPRING 4 SHELL

spring by VMware Tanzu

Why Spring | Learn | Projec

Spring Security Advisories

All advisories

CVE-2022-22965: Spring Framework RCE via Data Binding on JDK 9+

CRITICAL | MARCH 31, 2022 | CVE-2022-22965

Description

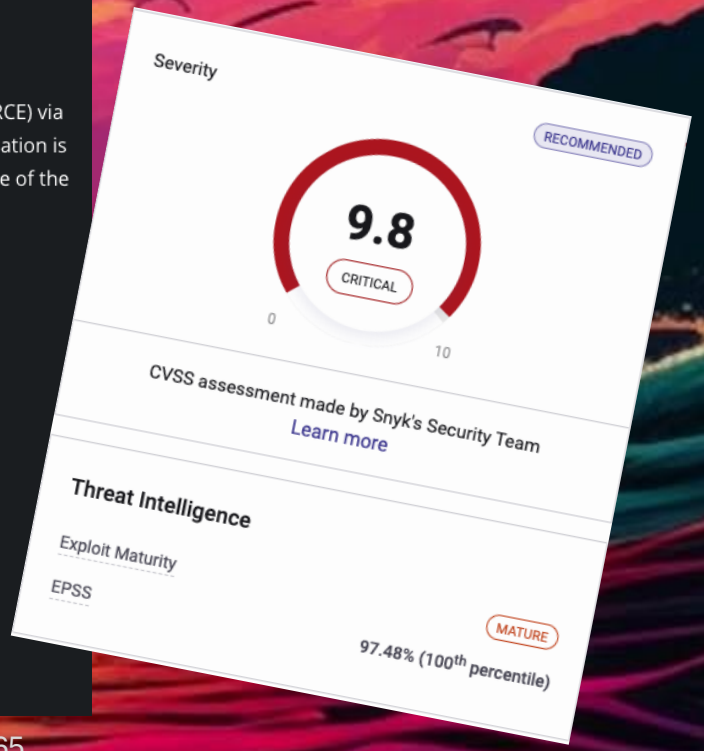
A Spring MVC or Spring WebFlux application running on JDK 9+ may be vulnerable to remote code execution (RCE) via data binding. The specific exploit requires the application to run on Tomcat as a WAR deployment. If the application is deployed as a Spring Boot executable jar, i.e. the default, it is not vulnerable to the exploit. However, the nature of the vulnerability is more general, and there may be other ways to exploit it.

These are the prerequisites for the exploit:

- JDK 9 or higher
- Apache Tomcat as the Servlet container
- Packaged as WAR
- spring-webmvc or spring-webflux dependency

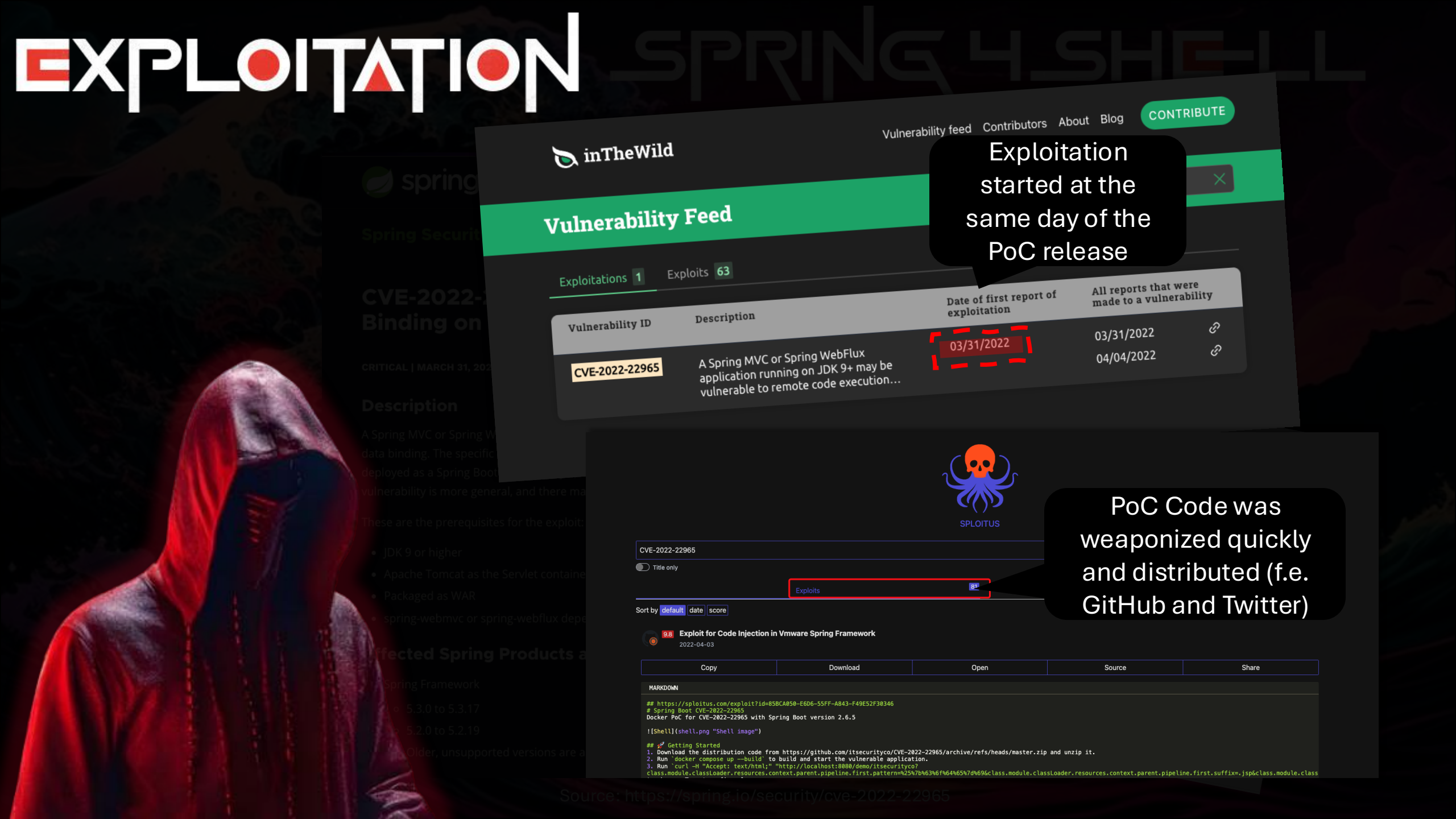
Affected Spring Products and Versions

- Spring Framework
  - 5.3.0 to 5.3.17
  - 5.2.0 to 5.2.19
  - Older, unsupported versions are also affected



Source: https://spring.io/security/cve-2022-22965

EXPLOITATION



Source: https://spring.io/security/cve-2022-22965

SPRING 4 SHELL

EXPLOITABILITY

1. JDK9 or above
2. Standalone Tomcat (no Embedded Tomcat) with WAR deployment
3. Any Springversion before 5.3.18/5.2.20 (SpringBoot before2.5.12 / 2.6.6)
4. No blocklist on WebDataBinder / InitBinder
5. Writeable filesystem (e.g. webapps / ROOT)
6. **Parameter bind with POJOs directly**
   (no @RequestBody, @RequestQuery, etc.)

Source: https://spring.io/security/cve-2022-22965

SPRING 4 SHELL

PoC Code to understand the Issue

Code | Blame | 23 lines (18 loc) · 686 Bytes

```java
package com.example.demo.controller;

import com.example.demo.model.EvalBean;
//import org.springframework.web.bind.WebDataBinder;
//import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class IndexController {

    @RequestMapping("/index")
    public void index(EvalBean evalBean) {

    }

    //@InitBinder
    //public void initBinder(WebDataBinder binder) {
    //    // mitigation:
    //    String[] blackList = { "class.*", "Class.*", "*.class.*", ".*Class.*" };
    //    binder.setDisallowedFields(blackList);
    //}
}
```

Source: https://github.com/DDuarte/springshell-rce-poc/blob/master/src/main/java/com/example/demo/controller/IndexController.java

SPRING 4 SHELL

Rule derived from PoC Code

```yaml
rules:
  - id: cve-2022-22965
    patterns:
      - pattern: "@$MAP(...) $R $M(..., $Y $X, ...) { ... }"
      # Classes and annotations that do not have the vulnerability based on:
      # https://docs.spring.io/spring-framework/docs/5.3.18/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleValueType-java.lang.Class-
      # https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-arguments
      - pattern-not: "@$MAP(...) $R $M(..., @CookieValue(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @MatrixVariable(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @PathVariable(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @RequestAttribute(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @RequestBody(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @RequestHeader(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @RequestParam(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @RequestPart(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @SessionAttribute(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., @SessionAttributes(...) $Y $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., String $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., boolean $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., byte $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., char $X, ...) { ... }"
      - pattern-not: "@$MAP(...) $R $M(..., short $X, ...) { ... }"
      - metavariable-pattern:
          metavariable: $MAP
          pattern-either:
            - pattern: RequestMapping
            - pattern: GetMapping
            - pattern: PostMapping
            - pattern: PutMapping
            - pattern: DeleteMapping
            - pattern: PatchMapping
      message: Semgrep found a match
      languages:
        - java
      severity: WARNING
      metadata:
        category: security
        cwe: "CWE-94: Improper Control of Generation of Code ('Code Injection')"
        owap: "A1: Injection"
        technology:
          - spring
```

Pattern for finding the issue

90 patterns to reduce noise conducted by not vulnerable code

Metavariables and context for the rule

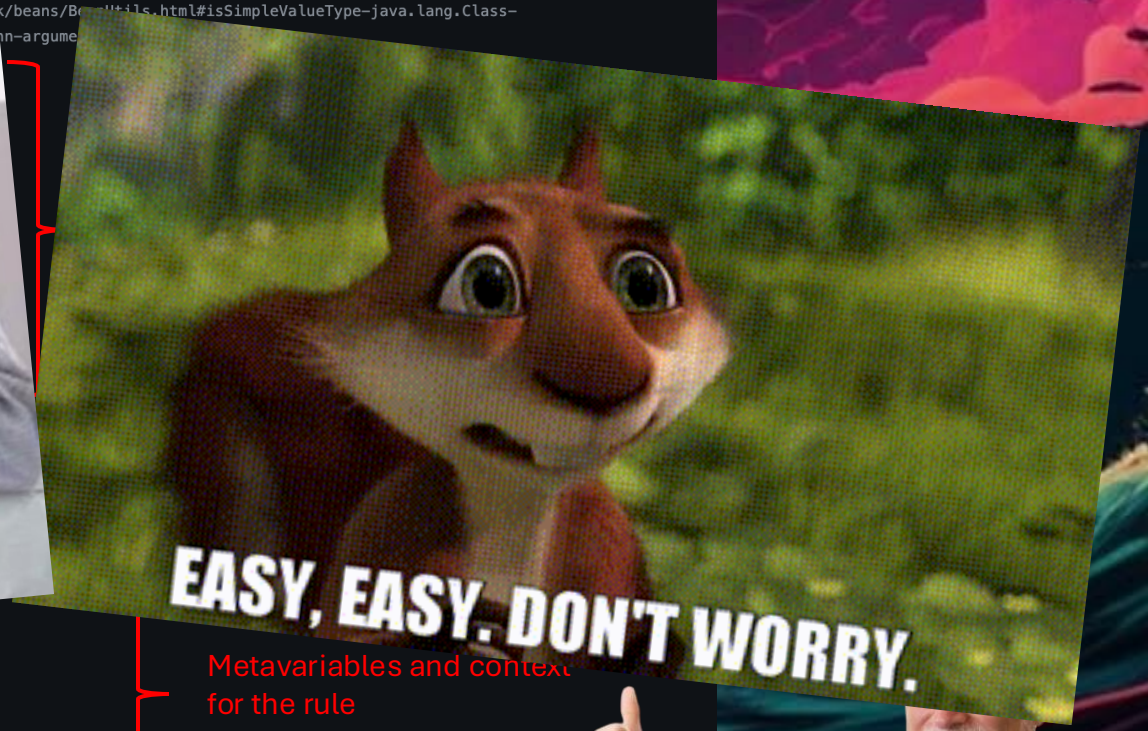
Source: https://github.com/DDuarte/springshell-rce-poc/blob/master/semgrep-rule.yml

SPRING 4 SHELL

Rule derived from PoC Code

```
1   rules:
2     - id: cve-2022-22965
3
```

for finding the issue




```
                - pattern: PatchMapping
108         message: Semgrep found a match
109         languages:
110           - java
111
112
113
114
115
116
117
```

Metavariables and context for the rule

To understand the magic behind a rule better, we learn it practical. All will be fine and don't forget to carry your towel!

Source: https://github.com/DDuarte/springshell-rce-poc/blob/master/semgrep-rule.yml

Lab2: A rule to find them and rule them all

Time estimation: 15min

Topics:

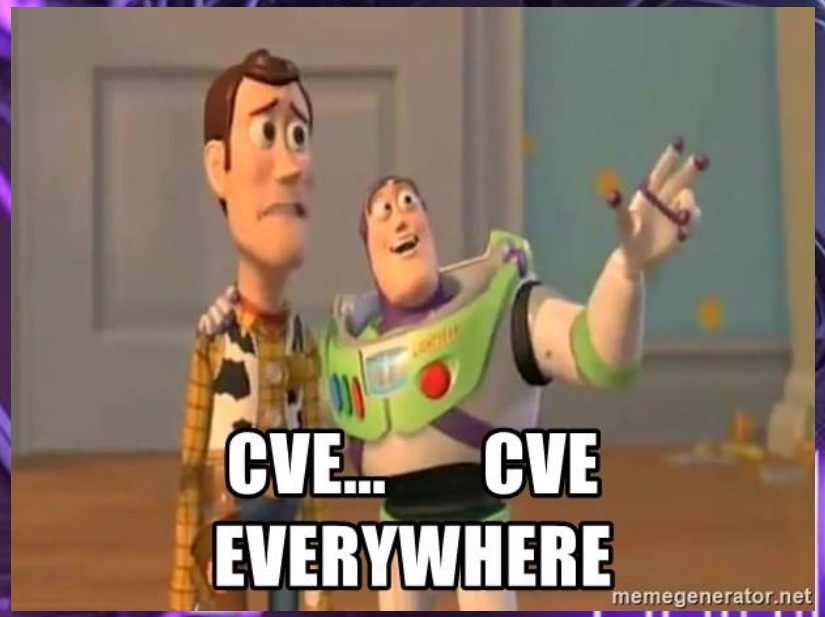
- Search Mode
- Writing the first rule
- Covering Business Logic
- IDE / Playground

What is

a CVE and a CWE

?

WHAT IS CVE?

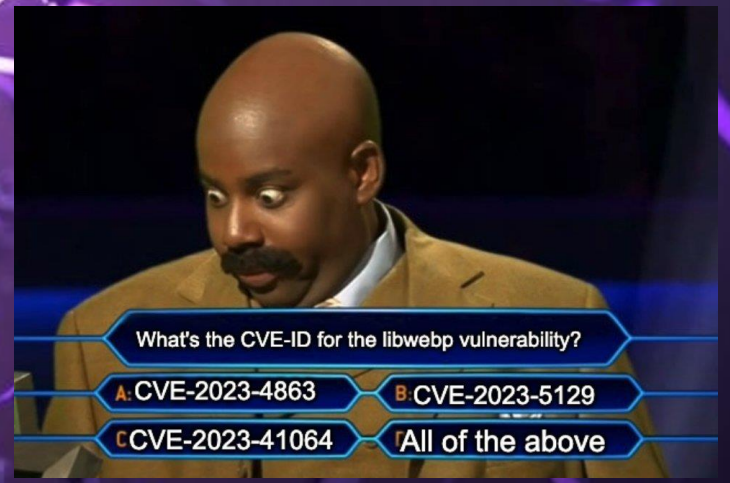


Common Vulnerabilities and Exposures (CVE) (also called "CVE names", "CVE numbers", "CVE-IDs", and "CVEs")

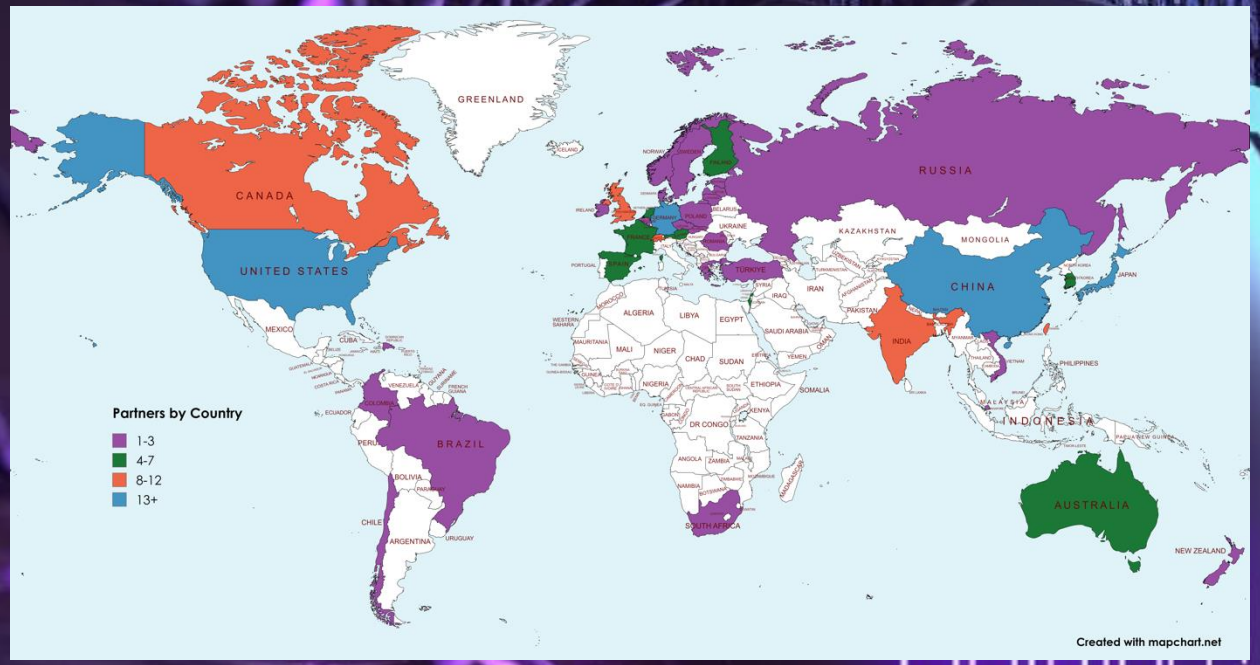
- Done by MITRE Corporation
- publicly released list of known computer security threats.
- Catalog of vulnerabilities in products.

Syntax of CVE

- **Structure:**
  - CVE prefix + Year + Arbitrary Digits
- **Variable Length Digits:**
  - Starts with four fixed digits
  - Expands with additional digits only as needed within a calendar year
  - Examples:
    - CVE-YYYY-NNNN
    - CVE-YYYY-NNNNN (if needed)
    - CVE-YYYY-NNNNNN (and so on)



CVE NUMBERING AUTHORITIES [CNAS]

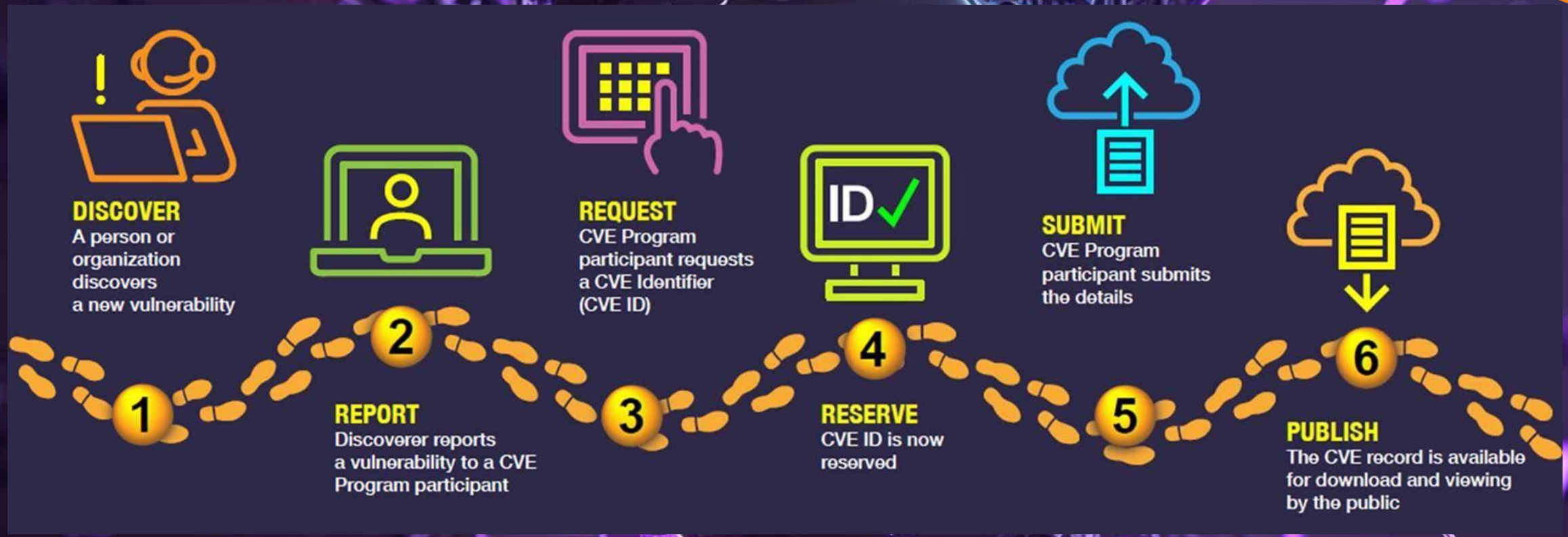


CVE identifiers (or CVE IDs) are assigned by a CVE Numbering Authority (CNA).

Currently, there are **414 CNAs** from 40 countries and 1 no country affiliation participating in the CVE Program, including security companies, research organizations, and major IT vendors such as Red Hat, IBM, Cisco, Oracle, and Microsoft. MITRE can also issue CVEs directly.

https://www.cve.org/ProgramOrganization/CNAs

HOW CVE IS BEING CREATED?



https://www.cve.org/about/Process

WHAT IS CWE?

- A “weakness” is a condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities.

- CWE (Common Weakness Enumeration) is a community-developed list of common software and hardware weakness types that could have security ramifications. Weakness conditions are in many cases introduced by the developer during development of the product.

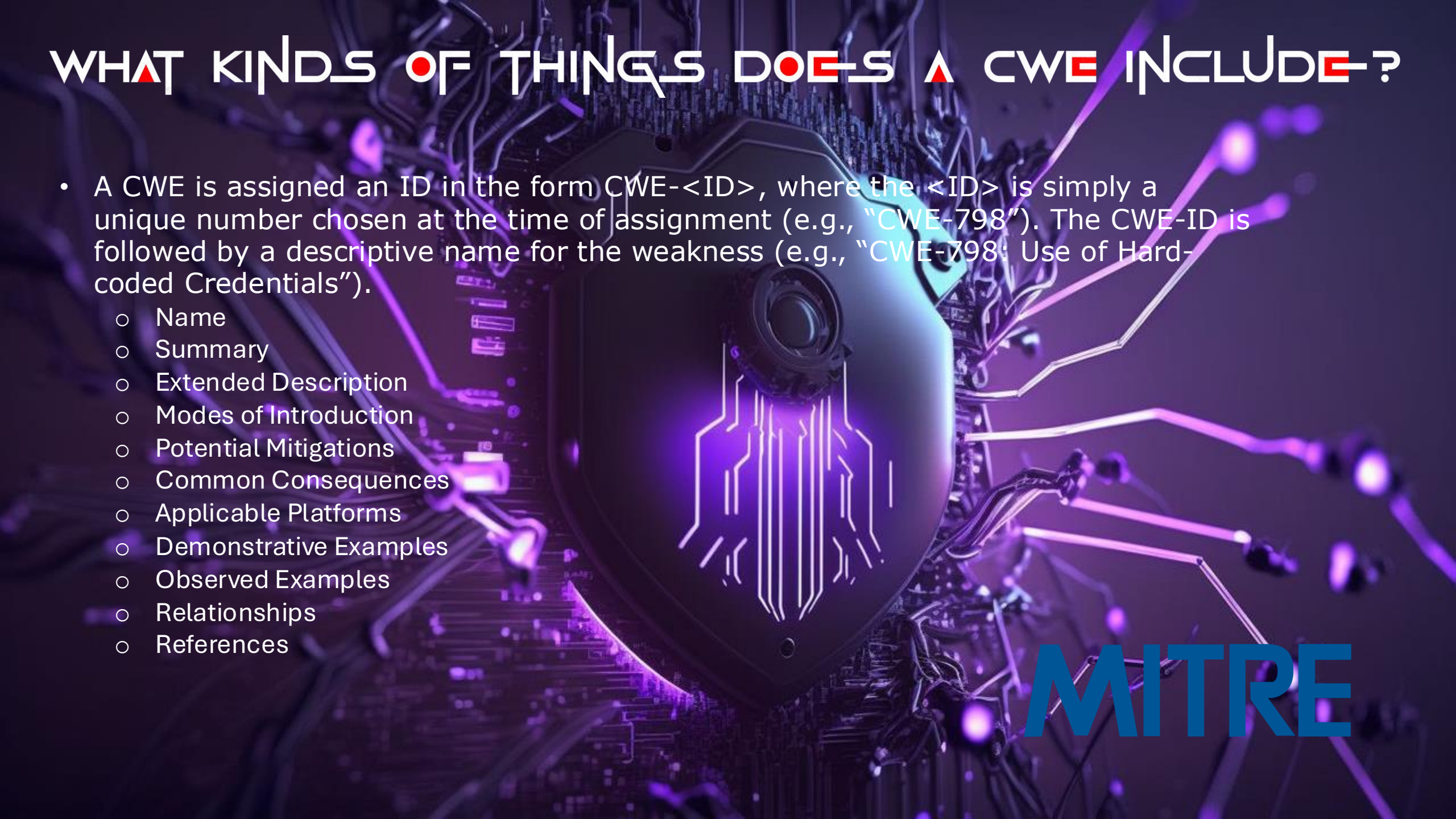
MITRE

WHAT IS CWE?

- Even though developers may have vastly different coding practices, they are all capable of introducing the same common type of weaknesses,
- The CWE List and associated taxonomies and classification schemes serve as a language that can be used to identify and describe these weaknesses in terms of “CWEs”.

What kinds of things does a CWE include?

- A CWE is assigned an ID in the form CWE-<ID>, where the <ID> is simply a unique number chosen at the time of assignment (e.g., "CWE-798"). The CWE-ID is followed by a descriptive name for the weakness (e.g., "CWE-798: Use of Hard-coded Credentials").
  - Name
  - Summary
  - Extended Description
  - Modes of Introduction
  - Potential Mitigations
  - Common Consequences
  - Applicable Platforms
  - Demonstrative Examples
  - Observed Examples
  - Relationships
  - References



BENEFITS OF CWE

- CWE allows developers to minimize weaknesses in their software as early in the lifecycle as possible, improving its overall security.

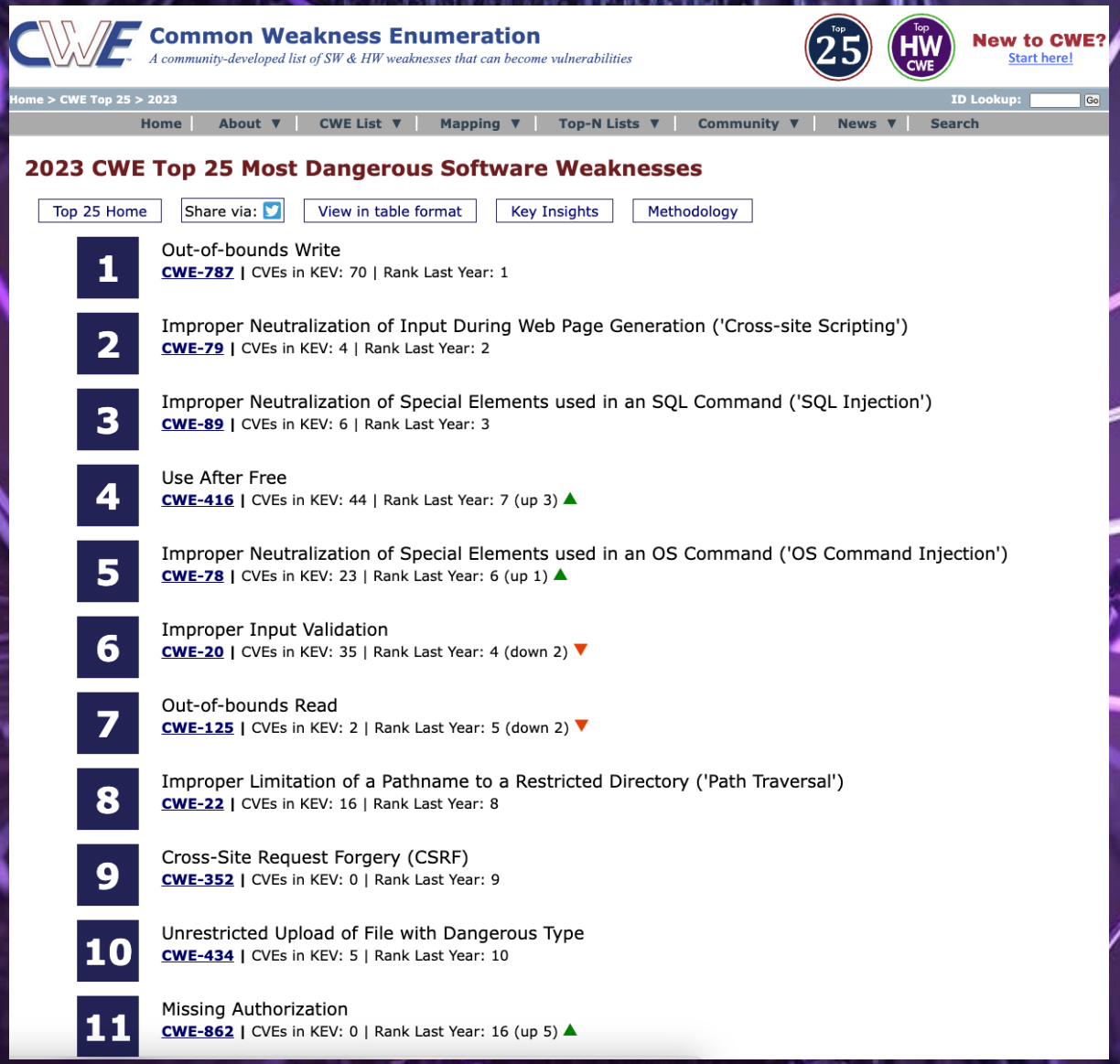
- CWE helps reduce risk industry-wide by enabling more effective community discussion about finding and mitigating these weaknesses in existing software and reducing them in future updates and releases.

- CWE enables more effective description, selection, and use of the software security tools and services that organizations can use to find these weaknesses and reduce their risk now.

Show me your SDLC I will say
How much secure it is

MITRE

2023 CWE TOP 25 MOST DANGEROUS SOFTWARE WEAKNESSES



Common Weakness Enumeration
A community-developed list of SW & HW weaknesses that can become vulnerabilities

New to CWE? Start here!

Home > CWE Top 25 > 2023 ID Lookup: Go

Home | About | CWE List | Mapping | Top-N Lists | Community | News | Search

2023 CWE Top 25 Most Dangerous Software Weaknesses

Top 25 Home | Share via: | View in table format | Key Insights | Methodology

1. Out-of-bounds Write
   CWE-787 | CVEs in KEV: 70 | Rank Last Year: 1
2. Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
   CWE-79 | CVEs in KEV: 4 | Rank Last Year: 2
3. Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
   CWE-89 | CVEs in KEV: 6 | Rank Last Year: 3
4. Use After Free
   CWE-416 | CVEs in KEV: 44 | Rank Last Year: 7 (up 3) ▲
5. Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
   CWE-78 | CVEs in KEV: 23 | Rank Last Year: 6 (up 1) ▲
6. Improper Input Validation
   CWE-20 | CVEs in KEV: 35 | Rank Last Year: 4 (down 2) ▼
7. Out-of-bounds Read
   CWE-125 | CVEs in KEV: 2 | Rank Last Year: 5 (down 2) ▼
8. Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
   CWE-22 | CVEs in KEV: 16 | Rank Last Year: 8
9. Cross-Site Request Forgery (CSRF)
   CWE-352 | CVEs in KEV: 0 | Rank Last Year: 9
10. Unrestricted Upload of File with Dangerous Type
    CWE-434 | CVEs in KEV: 5 | Rank Last Year: 10
11. Missing Authorization
    CWE-862 | CVEs in KEV: 0 | Rank Last Year: 16 (up 5) ▲

SOME EXAMPLES

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Weakness ID: 79
Vulnerability Mapping: ALLOWED
Abstraction: Base

View customized information: Conceptual | Operational | Mapping Friendly | **Complete** | Custom

Description

The product does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

Extended Description

Cross-site scripting (XSS) vulnerabilities occur when:

1. Untrusted data enters a web application, typically from a web request.
2. The web application dynamically generates a web page that contains this untrusted data.
3. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc.
4. A victim visits the generated web page through a web browser, which contains malicious script that was injected using the untrusted data.
5. Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malici
the context of the web server's domain.
6. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one
able to access resources or run code in a different domain.

There are three main kinds of XSS:

- **Type 1: Reflected XSS (or Non-Persistent)** - The server reads data directly from the HTTP request and refle
HTTP response. Reflected XSS exploits occur when an attacker causes a victim to supply dangerous content to a
application, which is then reflected back to the victim and executed by the web browser. The most common mecl
delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to
constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces a victim
refers to a vulnerable site. After the site reflects the attacker's content back to the victim, the content is executed
browser.
- **Type 2: Stored XSS (or Persistent)** - The application stores dangerous data in a database, message forum, visit
trusted data store. At a later time, the dangerous data is subsequently read back into the application and included i
content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed
users or particularly interesting users. Interesting users typically have elevated privileges in the application or intera
data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to pe
privileged operations on behalf of the user or gain access to sensitive data belonging to the user. For example, the att
inject XSS into a log message, which might not be handled properly when an administrator views the logs.
- **Type 0: DOM-Based XSS** - In DOM-based XSS, the client performs the injection of XSS into the page; in the other ty
server performs the injection. DOM-based XSS generally involves server-controlled, trusted script that is sent to the cli
Javascript that performs sanity checks on a form before the user submits it. If the server-supplied script processes user
data and then injects it back into the web page (such as with dynamic HTML), then DOM-based XSS is possible.

Once the malicious script is injected, the attacker can perform a variety of malicious activities. The attacker could transfer privat
information, such as cookies that may include session information, from the victim's machine to the attacker. The attacker could
malicious requests to a web site on behalf of the victim, which could be especially dangerous to the site if the victim has adminis
privileges to manage that site. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a pa
allowing the attacker to compromise the victim's account on that web site. Finally, the script could exploit a vulnerability in the web browser
itself possibly taking over the victim's machine, sometimes referred to as "drive-by hacking."

In many cases, the attack can be launched without the victim even being aware of it. Even with careful users, attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious.

Observed Examples

Reference	Description
CVE-2021-25926	Python Library Manager did not sufficiently neutralize a user-supplied search term, allowing reflected XSS.
CVE-2021-25963	Python-based e-commerce platform did not escape returned content on error pages, allowing for reflected Cross-Site Scripting attacks.
CVE-2021-1879	Universal XSS in mobile operating system, as exploited in the wild per CISA KEV.
CVE-2020-3580	Chain: improper input validation (CWE-20) in firewall product leads to XSS (CWE-79), as exploited in the wild per CISA KEV.
CVE-2014-8958	Admin GUI allows XSS through cookie.
CVE-2017-9764	Web stats program allows XSS through crafted HTTP header.
CVE-2014-5198	Web log analysis product allows XSS through crafted HTTP Referer header.
CVE-2008-5080	Chain: protection mechanism failure allows XSS
CVE-2006-4308	Chain: incomplete denylist (CWE-184) only checks "javascript:" tag, allowing XSS (CWE-79) using other tags
CVE-2007-5727	Chain: incomplete denylist (CWE-184) only removes SCRIPT tags, enabling XSS (CWE-79)
CVE-2008-5770	Reflected XSS using the PATH_INFO in a URL
CVE-2008-4730	Reflected XSS not properly handled when generating an error message
CVE-2008-5734	Reflected XSS sent through email message.
CVE-2008-0971	Stored XSS in a security product.
CVE-2008-5249	Stored XSS using a wiki page.
CVE-2006-3568	Stored XSS in a guestbook application.
CVE-2006-3211	Stored XSS in a guestbook application using a javascript: URI in a bbcode img tag.
CVE-2006-3295	Chain: library file is not protected against a direct request (CWE-425), leading to reflected XSS (CWE-79).

Weakness Ordinalities

WHY CWE IS IMPORTANT FOR SEMGREP?

Semgrep Registry Playground Products Pricing Docs turm

Library

e.g.: python.flask

- clojure 5
- csharp 52
- dockerfile 35
- elixir 7
- generic 257
- go 78
- html 6
- java 127
  - android 8
  - aws-lambda 2
  - java-jwt 3
  - jax-rs 3
  - jboss 2
  - jjwt 1
  - lang 88
  - mongodb 1
  - rmi 2
  - servlets 2
  - spring 15
  - security 15
    - audit 9
    - injection 5
      - tainted-file-path
      - tainted-html-string
      - tainted-sql-string
      - tainted-system-command
      - tainted-url-host
- unrestricted-request-mapping

tainted-sql-string

Fork Share Add to Policy

structure NEW | advanced

```yaml
rules:
  - id: tainted-sql-string
    languages:
      - java
    severity: ERROR
    message: User data flows into this manually-constructed SQL string. User data
      can be safely inserted into SQL strings using prepared statements or an
      object-relational mapper (ORM). Manually-constructed SQL strings is a
      possible indicator of SQL injection, which could let an attacker steal or
      manipulate data from the database. Instead, use prepared statements
      (`connection.PreparedStatement`) or a safe library.
    metadata:
      cwe:
        - "CWE-89: Improper Neutralization of Special Elements used in an SQL
          Command ('SQL Injection')"
      owasp:
        - A01:2017 - Injection
        - A03:2021 - Injection
      references:
        - https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html
      category: security
      technology:
        - spring
      license: Commons Clause License Condition v1.0[LGPL-2.1-only]
      cwe2022-top25: true
      cwe2021-top25: true
      subcategory:
        - vuln
      likelihood: HIGH
      impact: MEDIUM
      confidence: MEDIUM
      interfile: true
      vulnerability_class:
        - SQL Injection
    options:
      taint_assume_safe_numbers: true
      taint_assume_safe_booleans: true
      interfile: true
    mode: taint
    pattern-sources:
      - patterns:
```

test code | live code NEW | metadata | docs — Pro Turbo

```java
package com.r2c.tests;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.*;
import org.springframework.boot.autoconfigure.*;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;


@RestController
@EnableAutoConfiguration
public class TestController {

    private static final Logger LOGGER = LoggerFactory.getLogger(TestController.class);

    @RequestMapping(value = "/test1", method = RequestMethod.POST, produces = "plain/text")
    ResultSet test1(@RequestBody String name) {
        // ruleid: tainted-sql-string
        String sql = "SELECT * FROM table WHERE name = " + name + ";";
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:8080", "guest",
        "password");
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.execute(sql);
        return rs;
    }

    @RequestMapping(value = "/test2", method = RequestMethod.POST, produces = "plain/text")
    ResultSet test2(@RequestBody String name) {
        // ruleid: tainted-sql-string
        String sql = String.format("SELECT * FROM table WHERE name = %s;", name);
        Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:8080", "guest",
        "password");
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.execute(sql);
        return rs;
```

LAB3: TAINTED LOVE

Time estimation: 20min

Topics:

- Taint mode in action
- Exploring an example
- Learn about deserialization attacks
- Meeting the Playground


AND NOW FOR
SOMETHING
COMPLETELY
DIFFERENT
CLICK ME

What is

Left ....

?


NOW,
IT'S
YOUR
TURN

LAB4: TIME TO GET OUR HANDS DIRTY

Time estimation: 60min